FI MU

Faculty of Informatics
Masaryk University Brno

LTL model checking with I/O-Efficient Accepting Cycle Detection

by

Jiří Barnat
Luboš Brim
Pavel Šimeček

FI MU Report Series **FIMU-RS-2007-01**

 January 2007

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can obtained by contacting:

Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic

LTL model checking with I/O-Efficient Accepting Cycle Detection

JIŘÍ BARNAT, LUBOŠ BRIM, PAVEL ŠIMEČEK*
Faculty of Informatics
Masaryk University
Brno, Czech Republic
{barnat,brim,xsimece1}@fi.muni.cz

February 6, 2007

Abstract

We show how to adopt existing non-DFS-based algorithm OWCTY [ČP03] for accepting cycle detection to the I/O efficient setting and compare the I/O efficiency and practical performance of the adopted algorithm to the existing I/O efficient LTL model checking approach of Edelkamp et al. [EJ06]. We show that while the new algorithm exhibits similar I/O complexity with respect to the size of the graph, it avoids the quadratic increase in the size of the graph of the approach of Edelkamp et al. Therefore, the absolute numbers of I/O operations are significantly smaller and the algorithm exhibits better practical performance.

1 Introduction

LTL model checking [CGP99] became one of the standard technique for verification of hardware and software systems even though the class of the systems that can be fully verified, is fairly limited due to the well known *state explosion problem* [CGP99]. The commonly used automata-based approach to LTL model checking [Var96] reduces the problem of model checking to the problem of accepting cycle detection in a graph of a Büchi automaton. Due to the state explosion problem, the graph to be searched for the

*This work has been supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

presence of an accepting cycle tends to be extremely large. For that reason, the size of the graph poses real limitation to the verification process if performed on a single workstation. Many more-or-less successful reduction techniques have been introduced [CGP99] to fight the problem and to move the frontier of still tractable systems further. Nevertheless, for real-life industrial systems the reduction techniques are not efficient enough to make the verification tractable. A possible solution is to increase the computational resources available to the verification process. The two major approaches applied to increase the computational resources include the usage of clusters of workstations and the usage of external memory devices (disks).

Regarding external memory devices, the main limiting factor of the approach is the amount of I/O operations an algorithm has to perform to complete its task. This is because the access to information stored on the external device is in order of magnitude slower than the access to information stored in the main memory. Thus, it became common to measure the efficiency of an I/O algorithm in the number of I/O operations such as *random accesses*, *scans*, and *sorts* [AJ88].

A lot of effort has been put to research on I/O efficient algorithms working with explicitly stored graphs [KM02, MM02, KS96, CGG$^+$95]. For explicitly stored graph, an I/O algorithm has to perform a random access operation every time it needs to enumerate edges incident with a given vertex. However, for the model checking purposes, it is common that the graph is given implicitly meaning that the edges incident with a given vertex can be computed locally from the vertex proper. Thus, an algorithm working with implicit definition may save up to $|V|$ random access operations, which may have significant impact on the performance of the algorithm in practice.

A distinguished technique that allowed implementation of I/O efficient graph traversal procedures is the so called *delayed duplicate detection* [KS05, Kor04, SD98]. A traversal procedure has to maintain a set of visited vertices to prevent their re-exploration. Since the graphs are large, the set cannot be completely kept in the main memory and must be stored on the external memory device. When a new vertex is generated it is checked against the set to avoid its re-exploration. The idea of the delayed duplicate detection technique is to postpone the checks and perform them in a group for the price of a single scan operation instead of a bunch of random access operations.

Unfortunately, the delayed duplicate detection technique is incompatible with the depth-first search (DFS) of a graph [EJ06]. Therefore, most approaches to I/O efficient (LTL) model checking suggested so far, have focused on the state space genera-

tion and verification of safety properties only. The first I/O efficient algorithm for state space generation has been implemented in Murφ [SD98]. Later on several heuristics for the state space generation were suggested and implemented in various verification tools [KM03, HW06, JE05]. First attempt to verify more than safety properties was described in [JM04], however, the suggested approach uses the random search to find a counterexample to a given property, therefore, it is incomplete in the sense that it is not able to prove validity of the property.

To our best knowledge, the only complete I/O efficient LTL model checker was suggested in [EJ06] where the problematic DFS-based algorithm was avoided by the reduction of the accepting cycle detection problem to the reachability problem whose I/O efficient solution was further improved with the directed (A^*) search and parallelism. Nevertheless, the suggested reduction transforms the graph so that the size of the graph after the transformation is asymptotically quadratic with respect to the original size. More precisely, the resulting graph is of size $|F| \times |G|$, where $|G|$ is the size of the original graph and $|F|$ is the number of accepting vertices. As the I/O approach is meant to be applied first of all to large scale graphs, the quadratic increase in the size of the graph is significant and according to our experience often results in unsuccessful termination of the algorithm due to the lack of space. This is especially the case, if the model is valid and the graph has to be traversed completely to prove the absence of an accepting cycle. The approach is thus mainly useful for finding counterexamples in the case the standard verification tools fail to do so due to the lack of memory. The completeness of LTL model checking is, however, very important. A typical scenario is that if the system is invalid and the counterexample found, the system is corrected and the property verified again. In the end, the graph must be traversed completely anyway.

Since DFS-based algorithms cannot be used for I/O efficient solution to the accepting cycle detection, a non-DFS algorithm is required. The situation resembles the situation in the field of the cluster-based approach to LTL model checking [Bar04]. The main problem of the approach is that the standard algorithm Nested DFS [HPY96] is inherently sequential, so difficult to be parallelized [Rei85]. Fortunately, new parallel algorithms avoiding the problematic DFS have been introduced [BBS01, BvKP01, ČP03, BBC03, BČMŠ04]. In this paper we show how to adopt existing non-DFS-based from [ČP03] for accepting cycle detection to the I/O efficient setting and compare the I/O efficiency and practical performance of the algorithm to the existing I/O efficient LTL model checking approach of Edelkamp et al. [EJ06].

2 Algorithm

2.1 Algorithm of Černá and Pelánek [ČP03]

As discussed above, I/O efficient solution to LTL model checking has to build upon a non-DFS algorithm. A particularly interesting non-DFS algorithm for enumerative LTL model checking was described in [ČP03]. The idea of the algorithm is to compute the set of vertices that are reachable from an accepting cycle. If the set is empty, there is no accepting cycle in the graph, otherwise the presence of an accepting cycle is ensured.

The algorithm repeatedly computes approximations of the target set until a fixpoint is reached. All reachable vertices are inserted into the approximation set (*ApproxSet*) within the procedure INITIALIZE-APPROXSET. After that, vertices violating the condition are gradually removed from the approximation set using procedures ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS. Procedure ELIM-NO-ACCEPTING removes those vertices from the approximation set that have no accepting ancestors in the set, i.e. vertices that lie on leading non-accepting cycles. Procedure ELIM-NO-PREDECESSORS removes vertices that have no ancestors at all, i.e. leading vertices lying outside a cycle. The pseudo-code is given as Algorithm 1.

The approximation set induces an approximation graph. The in-degree of a vertex in the approximation graph corresponds to the number of its immediate predecessors in the approximation set. To identify vertices without ancestors in the approximation set, the in-degree is maintained for every vertex of the approximation graph. The procedure ELIM-NO-PREDECESSORS then works as follows. All vertices from the set with a zero in-degree are moved to a queue. Vertices are then dequeued one by one, eliminated from the set, and the in-degrees of its descendants updated. If an in-degree drops to zero, the corresponding vertex is inserted into the queue to be eliminated as well. The procedure eliminate vertices in a topological order, so the queue becomes empty as soon as all vertices preceding a cycle are eliminated.

Procedure ELIM-NO-ACCEPTING works as follows. If a vertex has an accepting ancestor in the approximation set, it has to be reachable from some accepting vertex in the set. In particular, the procedure first removes all non-accepting vertices from the set and sets the numbers of predecessors of all vertices remaining in the set to zero. Then a forward search is performed starting from the vertices remaining in the set. During the search all visited vertices are re-inserted to the approximation set and the numbers of immediate predecessors of vertices in the set are properly counted.

Algorithm 1 DETECTACCEPTINGCYCLE

Require: Implicit definition of G=(V,E,ACC)

```
1: INITIALIZE-APPROXSET()
2: oldSize ← ∞
3: while (ApproxSet.size ≠ oldSize) ∧ (ApproxSet.size > 0) do
4:     oldSize ← ApproxSet.size
5:     ELIM-NO-ACCEPTING()
6:     ELIM-NO-PREDECESSORS()
7: return ApproxSet.size > 0
```

2.2 I/O efficient Implementation

There are three major data structures used by the algorithm. These are *Candidates*, *ApproxSet*, and *Open*. *Candidates* is a set of vertices strictly kept in memory that is used for the delayed detection technique. It keeps vertices that have been processed and are waiting to be checked against the set of vertices stored on the external device. *ApproxSet* is a set of vertices belonging to the current approximation set. It is implemented as a linear list and stored externally. Together with *Candidates*, it is used as the set of already visited vertices during the forward exploration of the graph in procedure ELIM-NO-ACCEPTING. For that purpose, both *Candidates* and *ApproxSet* data structures are modified to keep not only vertices, but also the corresponding numbers of relevant immediate predecessors. The number associated with a particular vertex *s* is referred to as the *appendix* of the vertex and is set and read with methods *setAppendix(s)* and *getAppendix(s)*, respectively. Finally, data structure *Open* is a queue of vertices. It is used to keep open vertices during the breadth-first exploration of the graph within procedure ELIM-NO-ACCEPTING, and vertices to be eliminated (vertices with zero predecessors) during the execution of procedure ELIM-NO-PREDECESSORS. Data structure *Open* is stored in the external memory, however, vertices are inserted into and taken from the structure in a strict FIFO manner. Thus, a possible I/O overhead could be minimized using appropriate buffering mechanism.

In some phases, the algorithm performs a *scan* through the externally stored set of vertices (*ApproxSet*) and decides about every vertex if it should be removed from the set or not. To preserve the I/O efficiency of such an operation, a temporary external data structure *ApproxSet'* is introduced. In particular, vertices that should remain in the set are copied to the temporary structure. Once the scan is complete, the content of the

Algorithm 2 MERGE

```
1: if mode = Elim-No-Accepting then
2:     for all s ∈ ApproxSet do
3:         if s ∈ Candidates then
4:             app ← Candidates.getAppendix(s)
5:             app′ ← ApproxSet.getAppendix(s)
6:             Candidates ← Candidates \ {s}
7:             ApproxSet.setAppendix(s, app + app′)
8:     for all s ∈ Candidates do
9:         Open.pushBack(s)
10:        ApproxSet ← ApproxSet ∪ {s}
11: else
12:    ApproxSet′ ← ∅
13:    for all s ∈ ApproxSet do
14:        app′ ← ApproxSet.getAppendix(s)
15:        if s ∈ Candidates then
16:            app ← Candidates.getAppendix(s)
17:            if (app + app′) = 0 then
18:                Open.pushBack(s)
19:            else
20:                ApproxSet′ ← ApproxSet′ ∪ {s}
21:                ApproxSet′.setAppendix(s, app + app′)
22:        else
23:            ApproxSet′ ← ApproxSet′ ∪ {s}
24:            ApproxSet′.setAppendix(s, app′)
25:    ApproxSet ← ApproxSet′
26: Candidates ← ∅
```

original *ApproxSet* is discarded and replaced with the content of the temporary structure *ApproxSet′*.

Having described the data structures we are ready to introduce several auxiliary subroutines that the algorithm employs. The most important auxiliary procedure is procedure MERGE that is responsible for merging information about vertices stored in the memory (*Candidates*) and vertices stored externally (*ApproxSet*). The

procedure can operate in two different modes according to the value of the variable *mode*. The two modes correspond to the top most procedures ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS. In the mode **Elim-No-Accepting**, vertices from set *Candidates* are merged with vertices from *ApproxSet* and the result is stored externally to the set *ApproxSet*. For vertices visited before, only the corresponding appendices are combined and stored externally. Moreover, newly discovered vertices are inserted into the queue of vertices to be further processed (*Queue*). In the mode **Elim-No-Predecessors**, no new vertices are discovered, so only the appendices are combined. Vertices with zero in-degrees are removed from the external memory and are inserted in the queue so the in-degrees of their immediate descendants could be appropriately decreased. For details see the pseudo-code of Algorithm 2.

Another auxiliary procedure is procedure STOREORCOMBINE whose purpose is to insert a vertex into the candidate set if the vertex is not present in the set, or modify the corresponding appendix of the vertex, otherwise. New vertices are inserted into the set with the incoming appendices, for vertices already stored, both the stored and incoming appendices are combined and the originally stored appendix is replaced with the new combination. Once the main memory becomes full, vertices from the candidate set are processed and the candidate set is emptied by calling procedure MERGE.

Algorithm 3 STOREORCOMBINE

Require: s, app

```
1: if s ∈ Candidates then
2:     app' ← Candidates.getAppendix(s)
3:     Candidates.setAppendix(s, app+app')
4: else
5:     Candidates ← Candidates ∪ {s}
6:     Candidates.setAppendix(s, app)
7:     if MEMORYISFULL() then
8:         MERGE()
```

The last auxiliary function is a function to check the emptiness of the queue of vertices to be processed (*Open*). If the queue is empty, procedure OPENISNOTEMPTY calls procedure MERGE to perform the delayed duplicate detection. The procedure returns **False**, if *Open* is empty and the merging has not brought any new vertices to be processed.

Algorithm 4 OPENISNOTEMPTY

```
1: if Open.isEmpty() then
2:     MERGE()
3: return ¬Open.isEmpty()
```

Algorithm 5 ELIM-NO-ACCEPTING

```
1: mode ← Elim-No-Accepting
2: ApproxSet′ ← ∅
3: for all s ∈ ApproxSet do
4:     if ISACCEPTING(s) then
5:         Open.pushBack(s)
6:         ApproxSet′ ← ApproxSet′ ∪ {s}
7:         ApproxSet′.setAppendix(s, 0)
8: ApproxSet ← ApproxSet′
9: while OPENISNOTEMPTY() do
10:    s ← Open.popFront()
11:    for all t ∈ GETSUCCESSORS(s) do
12:        STOREORCOMBINE(t, 1)
```

Algorithm 6 ELIM-NO-PREDECESSORS

```
1: mode ← Elim-No-Predecessors
2: ApproxSet′ ← ∅
3: for all s ∈ ApproxSet do
4:     if ApproxSet.getAppendix(s) = 0 then
5:         Open.pushBack(s)
6:     else
7:         ApproxSet′ ← ApproxSet′ ∪ {s}
8: ApproxSet ← ApproxSet′
9: while OPENISNOTEMPTY() do
10:    s ← Open.popFront()
11:    for all t ∈ GETSUCCESSORS(s) do
12:        STOREORCOMBINE(t, −1)
```

Algorithm 7 INITIALIZE-APPROXSET

```
1:  mode ← Elim-No-Accepting
2:  Candidates ← ∅
3:  s ← GETINITIALVERTEX()
4:  ApproxSet ← {s}
5:  if ¬ ISACCEPTING(s) then
6:      Open.pushBack(s)
7:      while OPENISNOTEMPTY() do
8:          s ← Open.popFront()
9:          for all t ∈ GETSUCCESSORS(s) do
10:             if ISACCEPTING(t) then
11:                 ApproxSet ← ApproxSet ∪ {t}
12:             else
13:                 STOREORCOMBINE(t, 0)
```

Algorithm 5 and Algorithm 6 give pseudo-codes of the two main procedures. Note that the algorithm uses functions GETINITIALVERTEX, GETSUCCESSORS, and ISACCEPTING to traverse the graph and to check whether a vertex is accepting or not. These functions are part of the implicit definition of the graph. Procedure ELIM-NO-ACCEPTING has actually two goals. First, to eliminate vertices from the approximation set that are unreachable from the accepting vertices in the set, and second, to properly count the in-degrees in the approximation graph. Procedure ELIM-NO-PREDECESSORS employs the numbers of predecessors computed by procedure ELIM-NO-ACCEPTING to recursively remove vertices without predecessors from the approximation set. The procedure actually performs I/O efficient topological sorting.

An important observation is that it is not necessary to initialize the approximation set with all the vertices. Since the first procedure in the very first iteration of the while loop performs forward exploration of the graph starting from accepting vertices in the set, it is enough to initialize the set with "leading" accepting vertices only, i.e. those accepting vertices that have no accepting ancestors. Such vertices can be identified with a simple forward traversal that is allowed to explore descendants of non-accepting vertices only. See the pseudo-code given as Algorithm 7.

3 Complexity Analysis and Comparison

A widely accepted model for the analysis of the complexity of I/O algorithms is the model of Aggarwal and Vitter [AJ88], where the complexity of an I/O algorithm is measured in terms of the numbers of external I/O operations only. This is motivated by the fact that a single I/O operation is by approximately six orders of magnitude slower than a computation step performed in the main memory [Vit01]. Therefore, an algorithm that does not perform the optimal amount of work but has a lower I/O complexity, may be faster in practice compared to an algorithm that performs the optimal amount of work, but has a higher I/O complexity. The complexity of an I/O algorithm in the model model of Aggarwal and Vitter is further parametrized with M, B, and D, where M denotes the number of items that fits in the internal memory, B denotes the number of items that can be transferred in a single I/O operation, and D denotes the number of blocks that can be transferred in parallel, i.e. the number of independent parallel disks available. The abbreviations $\mathtt{sort}(n)$ and $\mathtt{scan}(n)$ stand for $\Theta(N/(DB)\log_{M/B}(N/B))$ and $\Theta(N/(DB))$, respectively. In this section we show the I/O complexity of our algorithm and compare it with the complexity of the algorithm from [EJ06].

3.1 I/O Complexity

The I/O complexity of our algorithm DETECTACCEPTINGCYCLE follows from the I/O complexities of functions INITIALIZE-APPROXSET, ELIM-NO-ACCEPTING, and ELIM-NO-PREDECESSORS.

To give the complexity precisely, we first remind several graph theory terms. *BFS tree* is a tree given by the graph traversal from the initial set of vertices in the breadth-first order. Its height is called *BFS height* and is denoted as h_{BFS}, its levels are called *BFS levels*. *SCC graph* is a directed acyclic graph, whose vertices are maximal strongly connected components of the graph and the edges are given according to the reachability relation between the components. Let l_{SCC} denote the length of the longest path in the SCC graph.

The complexities of individual procedures are given by the following lemmas starting with the complexity of the auxiliary function MERGE that is responsible for most I/O operations performed in all main procedures.

Lemma 3.1. *The I/O complexity of procedure* MERGE *is* $\mathcal{O}(\mathtt{scan}(|V|))$.

Proof. The loop on lines 2–7 of the pseudo-code scans the approximation set on the disk and updates appendices of revisited vertices. Each vertex is stored externally together with its appendix, so the access to the appendix (getAppendix, setAppendix) of the currently processed vertex does not produce any additional I/O operations. Since the approximation set keeps at most $|V|$ vertices, the loop performs $\mathcal{O}(scan(|V|))$ I/O operations. The loop on lines 13–24 behaves in a similar way, but instead of changing appendices of visited vertices, it appends every processed vertex to either *Open* or *ApproxSet'* data structures, which results in additional $\mathcal{O}(scan(|V|))$ I/O operations. Finally, the loop on lines 8–10 appends newly discovered vertices to both *Open* and *ApproxSet* data structures for the price of $\mathcal{O}(scan(|V|))$ I/O operations. Therefore, the total I/O complexity of the procedure is $\mathcal{O}(scan(|V|))$. □

Lemma 3.2. *The I/O complexity of* ELIM-NO-ACCEPTING *is* $\mathcal{O}((h_{BFS} + |E|/M) \cdot scan(|V|))$.

Proof. In the first part (lines 3–7 of the pseudo-code), the algorithm makes one scan through the entire approximation set and gradually creates its subset *ApproxSet'* and fills queue *Open* with its initial contents. This part of the algorithm writes and reads sequentially at most $2 \cdot |ApproxSet|$ data. Thus, the first part costs at most $\mathcal{O}(scan(|V|))$. The second part of the algorithm (lines 9–12) repeatedly merges *Candidates* with *ApproxSet* (the merging procedure is hidden in the functions OPENISNOTEMPTY and STOREORCOMBINE). The merge occurs every time the candidate set exhausts the internal memory or the queue *Open* becomes empty. It is guaranteed that each vertex is explored (i.e. its successors are generated) exactly once, because the vertex gets into the queue *Open* at the moment it is added to the approximation set (see line 9 in MERGE). Therefore, each vertex v is inserted into *Candidates* at most *in-degree*(v)-times. Consequently, the candidate set can consume the memory at most $(|E|/M)$-times. The queue becomes empty exactly once for each BFS level that is smaller than M. Therefore, the merge procedure is called at most $(h_{BFS} + |E|/M)$-times and the total I/O complexity of the procedure is $\mathcal{O}((h_{BFS} + |E|/M) \cdot scan(|V|))$. □

Lemma 3.3. *The I/O complexity of procedure* INITIALIZE-APPROXSET *is at most* $\mathcal{O}((h_{BFS} + |E|/M) \cdot scan(|V|))$.

Proof. The proof of the I/O complexity of procedure INITIALIZE-APPROXSET is similar to the proof of complexity of the procedure ELIM-NO-ACCEPTING. The procedure INITIALIZE-APPROXSET performs in essence the same state space traversal as procedure ELIM-NO-ACCEPTING. The only difference is that it might terminate earlier. □

Lemma 3.4. *The I/O complexity of procedure* ELIM-NO-PREDECESSORS *is* $\mathcal{O}((|p| + |E|/M) \cdot \mathrm{scan}(|V|))$, *where* p *is the longest path going through trivial strongly connected components (without self-loops) and starting from a vertex without predecessors in the approximation set.*

Proof. Let p be the longest path going through trivial strongly connected components and starting from a vertex without predecessors in the approximation set.

The procedure at first performs a single scan through the entire approximation set to divide the vertices between *ApproxSet'* and *Open* (lines 3–7 in the pseudo-code). This costs at most $\mathcal{O}(\mathrm{scan}(|V|))$ I/O operations. After that, *Open* contains all vertices without predecessors in the approximation set.

The second part of the algorithm (lines 9–12) repeatedly merges *Candidates* with *ApproxSet* (the merging procedure is hidden in the functions OPENISNOTEMPTY and STOREORCOMBINE). The merge occurs every time the candidate set exhausts the internal memory or the queue *Open* becomes empty.

As soon as all vertices from *Open* are processed, i.e. they are eliminated from the approximation set, the length of p must have been shortened. Otherwise, p was not the longest path satisfying the condition. After all vertices from path p are eliminated, there is no other vertex, that has no predecessor in the approximation set, therefore, the procedure terminates. Thus, the number of merge procedures called due to the emptiness of the queue *Open* is bounded by length of p.

Since each vertex is inserted into *Open* at most once, every edge is traversed at most once and consequently, every vertex v can be inserted into *Candidates* at most *in-degree*(v)-times. Thus, the candidate set can consume the memory at most $(|E|/M)$-times.

Therefore, within a single call to procedure ELIM-NO-PREDECESSORS, the merge procedure is called at most $(|p| + |E|/M)$-times and the I/O complexity of the procedure is $\mathcal{O}((|p| + |E|/M) \cdot \mathrm{scan}(|V|))$. □

Lemma 3.5. *The total I/O complexity of the algorithm* DETECTACCEPTINGCYCLE *is*

$$\mathcal{O}((l_{SCC} \cdot (h_{BFS} + |E|/M) + \textstyle\sum_i |p_i|) \cdot \mathrm{scan}(|V|)),$$

where p_i *is the longest path going through trivial strongly connected components (without self-loops) and starting from a vertex without predecessors in the approximation set during the* i*-th call of* ELIM-NO-PREDECESSORS.

Proof. The algorithm begins with execution of INITIALIZE-APPROXSET, which costs $\mathcal{O}((h_{BFS} + |E|/M) \cdot \mathrm{scan}(|V|))$. After that, the algorithm enters the main loop (lines 3–6).

The number of iterations of the external loop can be bounded with l_{SCC} [ČP03]. Therefore, procedure ELIM-NO-ACCEPTING brings in total $\mathcal{O}(l_{SCC} \cdot (h_{BFS} + |E|/M) \cdot scan(|V|))$.

A single call of ELIM-NO-PREDECESSORS costs $\mathcal{O}((|p_i| + |E|/M) \cdot scan(|V|))$. Then the total complexity for all calls of the procedure is $\mathcal{O}((\sum_i |p_i| + |E|/M) \cdot scan(|V|))$.

Together, the I/O complexity of the algorithm is in $\mathcal{O}((h_{BFS} + |E|/M) \cdot scan(|V|) + l_{SCC} \cdot (h_{BFS} + |E|/M) \cdot scan(|V|) + (\sum_i |p_i| + |E|/M) \cdot scan(|V|)) = \mathcal{O}((l_{SCC} \cdot (h_{BFS} + |E|/M) + \sum_i |p_i|) \cdot scan(|V|))$. □

Theorem 3.6. *The I/O complexity of algorithm* DETECTACCEPTINGCYCLE *is*

$$\mathcal{O}(l_{SCC} \cdot (h_{BFS} + |p_{max}| + |E|/M) \cdot scan(|V|)),$$

where p_{max} is the longest path in the graph going through trivial strongly connected components (without self-loops).

Proof. The theorem immediately follows from Lemma 3.5, since $l_{SCC} \cdot |p_{max}| \geq \sum_i |p_i|$. This result is a good compromise between accuracy of the upper complexity bound and comprehensibility of used notation. □

3.2 Comparison

For the purpose of comparison we refer to our new algorithm as to algorithm DAC and to the algorithm of Edelkamp et al [EJ06] as to algorithm EDL. [EJ06, Theorem 1] claims that with the algorithm EDL it is possible to detect accepting cycles with I/O complexity $\mathcal{O}(sort(|F||E|) + l \cdot scan(|F||V|))$, where $|F|$ is the number of accepting states and l is the length of the shortest counterexample.

The complexity of algorithm EDL is not easy to compare with our results, because the algorithms use different ways to maintain the set of candidates. The candidate set can be either stored externally (algorithm EDL) or internally (algorithm DAC). In the case that the candidate set is stored externally, it is possible to perform merge operation on a BFS level independently of the size of the main memory. Therefore, this approach is suitable for those cases where memory is small, or the graph is by the orders of magnitude larger. The disadvantage of the approach is that it needs sort operations and it cannot be combined with heuristics, such as bit-state hashing and a lossy hash table [HW06]. Fortunately, both algorithms EDL and DAC are modular enough to be able to work in both modes. Table 1 gives I/O complexities of all four variants, where

Candidate set in the main memory:

EDL′	$\mathcal{O}((l + \lvert F\rvert\lvert E\rvert/M) \cdot scan(\lvert F\rvert\lvert V\rvert))$
DAC	$\mathcal{O}(l_{SCC} \cdot (h_{BFS} + p_{max} + \lvert E\rvert/M) \cdot scan(\lvert V\rvert))$

Candidate set in the external memory:

EDL	$\mathcal{O}(l \cdot scan(\lvert F\rvert\lvert V\rvert) + sort(\lvert F\rvert\lvert E\rvert))$
DAC′	$\mathcal{O}(l_{SCC} \cdot ((h_{BFS} + p_{max}) \cdot scan(\lvert V\rvert) + sort(\lvert E\rvert)))$

Table 1: I/O complexity of algorithms for both modes of storage of the candidate set.

EDL′ denotes algorithm EDL modified so that the candidate set is kept in the internal memory, and DAC′ denotes algorithm DAC modified so that the candidate set is stored externally.

Note that for model checking graphs the numbers l_{SCC}, p_{max} and h_{BFS} are typically smaller by several orders of magnitude than the number of vertices. However, the number of accepting vertices (F) is typically in the same order of magnitude as the number of vertices. Therefore, EDL′ and EDL suffer from the graph blow-up and perform much more I/O operations compared to DAC and DAC′, respectively. On the other hand, EDL′ and EDL can outperform DAC and DAC′ on the graphs with small number of accepting vertices and short counterexamples.

3.3 Space Complexity

Regarding space complexity, the comparison is clear. Since algorithm EDL′ needs to remember all visited pairs of vertices, where the pair is made by an accepting vertex an arbitrary vertex, the space complexity of the algorithm is $\mathcal{O}(|F||V|)$, i.e. asymptotically quadratic in the size of the graph. On the other hand, the space complexity of algorithm DAC is $\mathcal{O}(|V|)$, as it only needs to maintain the approximation set, queue and the candidate set whose sizes are always bounded by the number of vertices. The same holds for the pair of algorithms EDL and DAC′.

4 Experimental evaluation

In order to obtain experimental evidence about how our algorithm behaves in practice, we implemented two verification tools and mutually compared their performance as well as we compared their performance with the performance of the popular model

Valid properties on large models.

		SPIN		LaS-BFS		Our algorithm	
	States	Time	RAM	Time	Disk	Time	Disk
Phils(16,1),P3	61,230,206	Out of memory		Out of memory		02:01:11	5.5 GB
MCS(5),P4	119,663,657	Out of memory		Out of memory		03:32:41	8 GB
Szymanski(5),P4	419,183,762	Out of memory		Out of memory		44:49:36	32 GB
Elevator2(16),P4	76,824,540	Out of memory		Out of memory		11:37:57	9.2 GB
Leader Fil.(7),P2	431,401,020	00:01:35	1369 MB	Out of memory		32:03:52	42 GB
Lamport(5),P4	76,824,540	00:00:59	469 MB	Out of memory		02:44:38	4.4 GB

Valid properties on small models.

		SPIN		LaS-BFS		Our algorithm	
	States	Time	RAM	Time	Disk	Time	Disk
Lamport(3),P4	56,377	00:00:01	18 MB	00:55:34	799 MB	00:00:19	6,1 MB
Anderson(4),P2	58,205	00:00:01	20 MB	00:11:11	153 MB	00:00:18	6,1 MB
Peterson(4),P4	2,239,039	00:00:08	85 MB	Out of memory		00:04:44	159 MB

Invalid properties.

		SPIN		LaS-BFS		Our algorithm	
	States	Time	RAM	Time	Disk	Time	Disk
Bakery(5,5),P3	506,246,410	00:00:01	16 MB	01:34:13	5,4 GB	69:27:58	38 GB
Szymanski(4),P2	4,555,287	00:00:01	18 MB	00:59:00	203 MB	00:19:55	205 MB
Elevator2(7),P5	43,776	00:00:01	17 MB	00:01:15	121 MB	00:00:18	6,1 MB

Table 2: Run times and memory consumption on a single workstation with 2 GB of RAM and 60 GB of available hard disk space. The time is given in `hh:mm:ss` format.

checker SPIN with all the default reduction techniques (including partial order) turned on.

Regarding our algorithm, we implemented procedure *DetectAcceptingCycle* (DAC) upon DiVinE Library [BBvv05] providing the state space generator, and STXXL Library [DKS05] providing the necessary I/O primitives. As for the algorithm of Edelkamp et al, we implemented a procedure that performs the graph transformation as suggested in [EJ06] and then employs I/O efficient breadth-first search to check for the counterexample. Note that our implementation of [EJ06] does not have the A^* heuristics and so it can be less efficient in the search for the counterexample when it is present. The procedure is referred to as *Liveness as Safety with BFS* (LaS-BFS).

We have measured run times and a memory consumption of SPIN, LaS-BFS and DAC on a collection of systems and their LTL properties taken from BEEM

project [Pel07]. Note that since DiVinE and SPIN have different input languages, we take special care of the equivalency of the models we used during the experimental evaluation. The models were selected so that the state spaces generated by SPIN and DiVinE were exactly the same size. As all used models are parametrized, the equality of state spaces has been verified using smaller instances of used models [1]. The experimental results are listed in Table 2.

Measurements on large systems with valid formulas demonstrate that DAC is able to successfully prove the correctness of systems, on which SPIN and LaS-BFS run out of memory. However, there are systems and valid formulas, which take a long time to verify by our algorithm, but can be verified quickly using SPIN (e. g. model *Leader Filters*). It is due to the partial order reduction technique, which is extraordinarily efficient in this case. Results on small systems show the state space blow-up in case of LaS-BFS. E. g. on model *Lamport*, 6,1 MB of disk space is enough for DAC to store the entire state space, but LaS-BFS needs 799 MB. As for systems with invalid formulas, the new algorithm is slow, since it is not on-the-fly. Nevertheless, it is able to finish if the state space fits in the external memory. Moreover, it is faster than LaS-BFS on systems with long counterexamples as the space space blow-up takes effect when LaS-BFS has to traverse a substantial part of the state space (e. g. model *Elevator2*).

In summary, the new algorithm is especially useful for verification of large systems with valid formulas where SPIN fails due to the limited size of the main memory and LaS-BFS runs out of the available external memory because of a large amount of accepting states. On systems with invalid formulas it finishes if the state space fits in the external memory, but it may take quite a long time, since the algorithm does not work on-the-fly.

5 Conclusions

In this paper we presented a new I/O efficient algorithm for accepting cycle detection on implicitly given graphs. To our best knowledge, the algorithm is the first algorithm that exhibits linear space complexity while preserving practically reasonable I/O complexity. A distinguished contribution of the paper is that we were also the first

[1] State space sizes of Promela models were measured using commands `spin -a -o2 model.pm; gcc -O2 -DNOREDUCE pan.c; ./a.out -E -m1000000 -w22`

who introduced I/O efficient topological sorting on implicitly given graphs (procedure ELIM-NO-PREDECESSORS).

Our experimental evaluation confirmed that the new I/O algorithm is able to fully solve instances of the LTL model checking problem that cannot be solved either with the standard LTL model checker SPIN or using so far the best I/O efficient approach of Edelkamp et al [EJ06]. The approach of Edelkamp et al fails especially if the verified formula is valid, which is because after the transformation, the graph becomes too large to be kept even in the external memory.

On the other hand, unlike SPIN and the approach of Edelkamp et al, our algorithm does not work on-the-fly. The on-the-fly algorithms are particularly successful if the property is violated and the counterexample can be found early during the state space exploration.

5.1 Future work

As our algorithm is based on the algorithm which can be easily parallelized [ČP03], it should be easy to develop a parallel version of our algorithm and thus, get the algorithm for verification of yet larger systems. It also seems promising to make other BFS-based verification algorithms I/O efficient [BBS01, BvKP01, BBC03, BČMŠ04]. Some of them are on-the-fly and so they could outperform both, our new algorithm and the algorithm of Edelkamp et al [EJ06].

An open problem for which we still do not know a practically good solution, is the inefficiency of the delayed duplicate detection technique on graphs with the big BFS height and small BFS levels. Since the merge procedure taking several minutes on reasonably large graphs is called after every BFS level traversal, the speed of the exploration falls down to few vertices per minute. It would be sometimes more efficient to generate several BFS levels and remove all duplicates at once. Nevertheless, it is not easy to estimate a number of levels to generate at once, since a computing time needed for exploration of duplicate vertices can surpass the time saved by omitting several merge operations.

References

[AJ88] Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[Bar04] Jiří Barnat. *Distributed Memory LTL Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University Brno, 2004.

[BBC03] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 106–115. IEEE Computer Society, Oct. 2003.

[BBS01] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL Model-Checking in SPIN. In *Proc. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 200 – 216. Springer, 2001.

[BBvv05] J. Barnat, L. Brim, I. Černá, and P. Šimeček. Divine - the distributed verification environment. In M. Leucker and J. van de Pol, editors, *Proc. of 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC05)*, pages 89–94, 2005.

[BČMŠ04] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed ltl model-checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.

[BvKP01] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In *Proc. of Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[ČP03] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In T. Ball and S.K. Rajamani, editors, *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 49 – 73. Springer Verlag, 2003.

[DKS05] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard template library for XXL data sets. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Algorithms - ESA 2005 : 13th Annual European Symposium*, volume 3669 of *Lecture Notes in Computer Science*, pages 640–651, Palma de Mallorca, Spain, 2005. EATCS, Springer.

[EJ06] Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking ltl. In *SPIN*, pages 1–18, 2006.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The SPIN Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the 2nd SPIN Workshop.

[HW06] Moritz Hammer and Michael Weber. To store or not to store reloaded: Reclaiming memory on demand. In *FMICS*, 2006.

[JE05] Shahid Jabbar and Stefan Edelkamp. I/o efficient directed model checking. In *VMCAI*, pages 313–329, 2005.

[JM04] Michael Jones and Eric Mercer. Explicit state model checking with hopper. In *SPIN*, pages 146–150, 2004.

[KM02] Irit Katriel and Ulrich Meyer. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*, pages 62–84, 2002.

[KM03] Lars Michael Kristensen and Thomas Mailund. Efficient path finding with the sweep-line method using external storage. In *ICFEM*, pages 319–337, 2003.

[Kor04] Richard E. Korf. Best-first frontier search with delayed duplicate detection. In *AAAI*, pages 650–657, 2004.

[KS96] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, page 169, Washington, DC, USA, 1996. IEEE Computer Society.

[KS05] Richard E. Korf and Peter Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.

[MM02] K. Mehlhorn and U. Meyer. External-memory breadthfirst search with sublinear i/o. In *l0th Annual European Symposium on Algorithms*, pages 723–735. Springer, 2002.

[Pel07] R. Pelánek. BEEM: BEnchmarks for Explicit Model checkers. `http://anna.fi.muni.cz/models/index.html`, February 2007.

[Rei85] J.H. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[SD98] U. Stern and D. L. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *Computer Aided Verification. 10th International Conference*, pages 172–183, 1998.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 238–266, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, 2001.